TD6 : programmation orientée objet

I. Notion d'attributs et de classes

Certains objets ont des propriétés que d'autres objets n'ont pas. Par exemple les nombres ont une partie réelle et une partie imaginaire, alors les mots n'en ont pas :

```
a=3; b=-7.2; c="3"
for z in [a,b,c]:
    print("la partie reelle de "+str(z)+" est "+str(z.real)+
                " et sa partie imaginaire est "+str(z.imag))
```

En python, de telles propriétés s'appellent des attributs et sont spécifiques à des types d'objets précis.

On accède à ses attributs en utilisant le symbole "`.`" : par exemple la partie de réelle de z est accessible par la commande "`z.real`".

Pourquoi python renvoit-il un message d'erreur en exécutant le code ci-dessus ?

Les attributs dont dispose un objet sont fixé par le type de l'objet. Par exemple les chaînes de caractères n'ont pas de partie réelle, d'où le message d'erreur en exécutant la commande ci-dessous :

```
c.real
```

On qualifie souvent de "classes" les différents types d'objets (auxquels sont attachés des attributs).

Méthodes

Certains attributs sont des fonctions, comme ci-dessous, où "`l.append(7)`" ajoute l'élément "7" à la liste `l`.

```
L=[]
L.append(7)
print(L)
```

Ces attributs qui sont des fonctions s'appellent des "méthodes".

Argument "self"

Cette méthode `append` existe car `L` est une liste, c'est pourquoi cette fonction s'appelle aussi `list.append`.

Quand on l'utilise sous la forme `list.append`, il est nécessaire d'indiquer comme premier argument le nom de la liste, comme ci-dessous :

```
list.append(L,12)
print(L)
```

Plus généralement, chaque méthode est une fonction qui prend un ou plusieurs arguments (par exemple la méthode `list.append` prend deux arguments).

Le premier argument s'appelle conventionnellement `self` et désigne l'objet auquel s'applique cette méthode.

Quand on exécute `L.append(7)` python comprend d'une part qu'`append` désigne la méthode `list.append`, et d'autre part que le premier argument de cette fonction doit être remplacé par `L` et le second par `7`.

Méthodes spéciales

Parmi ces méthodes, certaines sont particulièrement importantes, comme les méthodes `__init__`, `__repr__`, etc. dont nous parlerons plus loin.

```
print('par exemple, "L.__repr__()" renvoit la chaîne de caractère :"'+
                                                    L.__repr__()+'"')
```

II. Exemple de classe : grandeurs physiques

Les grandeurs physiques sont des nombres potentiellement munis d'unité. Toute unité s'écrit comme produit des unités fondamentales suivantes : le kilogramme, la seconde, le mètre, le Kelvin, l'Ampère, la mole et le candela (chacune de ces unités pouvant être élevée à une puissance quelconque).

Par exemple la vitesse "50 km/h" vaut environ $13.88\,m\,s^{-1}$.

On définit ci-dessous une classe

```
class grandeur:
    units=["kg","s","m","K","A","mol","cd"] #liste des unités
    def __init__(self,val,pow=[0]*len(units)):
        self.valeur=val
        self.puissances=pow
```

```
    def __repr__(self):
        res=str(self.valeur)
        for i in range(len(grandeur.units)):
          if self.puissances[i]!=0:
            res=res+"*"+grandeur.units[i]+"**"+str(self.puissances[i])
        return res
    def memeunite(self,B):#vérifie si deux grandeurs ont mêmes unités
        return self.puissances==B.puissances

A=grandeur(12,[0,0,1,0,0,0,0])
B=grandeur(3,[1,0,0,0,0,0,0])
print(A)
print(B)
print(A.memeunite(B))
```

Explications :

- On définit tout d'abord un objet `units`.
  Comme il est définit à l'intérieur de la classe `grandeur`, on y accède sous le nom `grandeur.units`.
  Il contient la liste des unités dont est constituée une grandeur physique.
- On construit ensuite une méthode `__init__`.
  Cette méthode est appelée à chaque fois qu'on crée un objet de type `grandeur`. Par exemple, quand on exécute `A=grandeur(12,[0,0,1,0,0,0,0])`, la fonction `__init__` est appelée, et
  - `self` désigne alors la grandeur physique créée (c'est à dire `A`)
  - L'argument `val` vaut `12`, tandis que l'argument `pow` vaut `[0,0,1,0,0,0,0]`.

  En conséquence la commande `self.valeur=val` enregistre la valeur de `val` dans un attribut `A.valeur`. (de même la commande `self.puissances=pow` enregistre le dernier argument dans un attribut `puissances`).
  Ainsi la commande `A=grandeur(12,[0,0,1,0,0,0,0])` a simplement créé un objet `A` dont l'attribut `A.valeur` vaut `12` et l'attribut `A.puissances` vaut `[0,0,1,0,0,0,0]`.
- La méthode `__repr__` définit la façon dont s'affiche une grandeur physique, par exemple lorsqu'on demande `print(A)`.
  - Cette fonction commence par considérer la valeur numérique (l'attribut `self.valeur`).
  - Elle ajoute ensuite ensuite le nom de chaque unité qui apparaît (avec une puissance non nulle) dans la grandeur physique.
- Enfin, la méthode "`memeunite`" permet de savoir si deux grandeurs physique ont, ou pas, la même unité (ce qui permet par exemple de les additionner).

1. Méthodes reconnues par python pour faire des calculs

Créer des méthodes `__add__`, `__sub__`, `__mul__`, etc permet de dire à python comment faire des calculs avec le objets que l'on a définis. Par exemple quand on demande à python de calculer `A+B`, il exécute en fait `A.__add__(B)`. Ainsi, définir une méthode `__add__` indique à python comment faire une addition.

Ci-dessous, on a donc ajouté d'autres méthodes à la classe `grandeur` (et modifié un peu `__init__` et `__repr__`) :

```
class grandeur:
    units=["kg","s","m","K","A","mol","cd"] #liste des unités
    def __init__(self,val,pow=[0]*len(units)):
        self.valeur=val
        if type(pow)!=list: #permet de définir l'unité par du texte,
              #par exemple en définissant 1 km par grandeur(1000,"m")
            if pow in grandeur.units:
                pow=[1 if u==pow else 0 for u in grandeur.units]
            else: raise ValueError("unité non identifiée")
        self.puissances=pow
    def __repr__(self):
        res=str(self.valeur)
        for i in range(len(grandeur.units)):
            if self.puissances[i]>0:
                res=res+" "+grandeur.units[i]
                if self.puissances[i]!=1:
                    res=res+"**"+str(self.puissances[i])
        for i in range(len(grandeur.units)):
            if self.puissances[i]<0:
                res=res+"/"+grandeur.units[i]
                if self.puissances[i]!=-1:
                    res=res+"**"+str(-self.puissances[i])
        return res
    def memeunite(self,B):#vérifie si deux grandeurs ont mêmes unités
        return self.puissances==B.puissances
    def __add__(self,B):#addition: calcul de self+B
        if self.memeunite(B):
```

```
            return grandeur(self.valeur+B.valeur,self.puissances)
        else: raise ValueError("""On ne peut pas ajouter des
                    grandeurs physiques qui n'ont pas la même unité""")
    def __mul__(self,B):#multiplication: calcul de self*B
        return grandeur(self.valeur*B.valeur,[
self.puissances[i]+B.puissances[i] for i in range(len(grandeur.units))])
    def __pow__(self,n): #power: calcul de self**n
        return grandeur(....)
    def __div__(self,B):# division: calcul de self/B en python 2
        return self*(B**(-1))
grandeur.__truediv__=grandeur.__div__ #définit la division pour python 3
```

Compléter la définition de l'attribut __pow__ qui calcule une puissance d'une grandeur physique.

On peut dès lors exprimer, par exemple la grandeur $50\,km/h$:

```
km=grandeur(1000,"m")
min=grandeur(60,"s")
h=grandeur(60)*min
print(grandeur(50)*km/h)
```

III. Polynômes à coefficients réels

Définir une classe qui permette de manipuler des polynômes.
En particulier, celle-ci permettra de calculer le pgcd de deux polynômes, par l'algorithme d'Euclide

```
class polynome:
    def __init__(self,coefficients,precision=10**-10):
        """ initialise un polynome à partir de la liste de ses coefficients.
        si un coeffiient est plus petit que "precision", il sera considéré comme nul (issu d'une "erreur d'arrondi")"""
        self.coeffs=[(float(i) if abs(i)>precision else 0) for i in coefficients]
        if all(c==0 for c in self.coeffs): self.degre=-1
        else: self.degre=max(i for i in range(len(self.coeffs)) if self.coeffs[i]!=0)
        self.coeffs=self.coeffs[:self.degre+1] #supprime d'eventuels coefficients nuls
    def __repr__(self):
        res=""
        for i in range(len(self.coeffs)):
            if self.coeffs[i]!=0:
                if res!="" and self.coeffs[i]>0: res=res+"+"
                if self.coeffs[i]==-1 and i!=0: res=res+"-"
                elif self.coeffs[i]!=1 or i==0:
                    res=res+str(self.coeffs[i])
                if i>0: res=res+"X"
                if i>1: res=res+"**"+str(i)
        if res=="": return "0"
        return res
    def coef(self,n): return self.coeffs[n] if n<= self.degre else 0
    def coef_dominant(self):
        return self.coeffs[-1] if self.degre>=0 else 0
    def __add__(self,B):#addition
        ...
    def __sub__(self,B):#soustraction
        ...
    def __mul__(self,B):#multiplication
        ...
    def __mod__(self,B): #reste de division euclidienne
        ...
    def __floordiv__(self,B):#quotient de division euclidienne
        ...
    def pgcd(self,B):
        ...
```